Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Jakofčič

Načrtovanje računalnika od logičnih vrat do visokonivojskega programskega jezika

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: prof. dr. Branko Šter

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Knjiga The Elements of Computing Systems opisuje izdelavo preprostega računalnika od najmanjših gradnikov (logičnih vrat) do operacijskega sistema in visokonivojskega programskega jezika. S tem bralcu omogoči na ilustrativnem zgledu vpogled v delovanje računalnika na različnih nivojih. Opišite načrtovanje računalnika po tej knjigi. Na koncu razširite arhitekturo računalnika z dodanima ukazoma za množenje in deljenje.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Domen Jakofčič sem avtor diplomskega dela z naslovom:

Načrtovanje računalnika od logičnih vrat do visokonivojskega programskega jezika

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Branka Štera,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 30. september 2017 Podpis avtorja:

Za pomoč pri izdelvi diplomskega dela se zahvaljujem prof. dr. Branku Šteru za mentorstvo in koristne nasvete. Hvala tudi družini in punci za podporo v času študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
HDL	hardware description language	jezik za opis strojne opreme
API	application programming interface	aplikacijski programski vmesnik
DFF	D flip-flop	pomnilna celica D
RAM	random access memory	bralno pisalni pomnilnik
CPU	central processing unit	centralna procesna enota **(CPE)**
ALU	arithmetic-logic unit	aritmetično-logična enota **(ALE)**
ASCII	American standard code for information interchange	standardna koda za zapis znakov
ROM	read only memory	bralni pomnilnik

Povzetek

Naslov: Načrtovanje računalnika od logičnih vrat do visokonivojskega programskega jezika

Cilj diplomskega dela je po knjigi The Elements of Computing Systems [1] izdelati enostaven računalnik, na katerem bo možna uporaba objektno usmerjenega programskega jezika Jack. Ta je zasnovan v sklopu implementacije računalnika. Pri izdelavi se srečamo s temami, kot so strojna oprema, arhitektura, podatkovne strukture, algoritmi, programski jeziki, prevajalniki, operacijski sistemi in razvoj programske opreme. Vsako izmed področij pokriva zelo širok del računalništva, kar velikokrat predstavlja problem pri razumevanju delovanja računalnika kot celote. Pri tem svojo vlogo odigra knjiga [1], ki v grobem predstavi vsako izmed tem, ter s pomočjo nalog omogoča razumevanje dogajanja v računalniku od prevajanja visokonivojskega programskega jezika v strojno kodo do izdelave strojne opreme, na kateri jo je možno zaganjati.

Ključne besede: računalnik, strojna oprema, programska oprema, operacijski sistem, centralna procesna enota, prevajalnik, navidezni stroj.

Abstract

Title: Designing a Computer from Logic Gates to a High-Level Programming Language

The objective of this diploma thesis is to build a simple computer, which will enable the use of the Jack object-oriented programming language, by following instructions from the book The Elements of Computing Systems [1]. This programming language has been designed as a part of the implementation of the computer. While building the computer, the following topics are encountered: hardware, architecture, data structures, algorithms, programming languages, compilers, operating systems and software development. Each of these topics covers a very broad area of computer science, which often poses a problem when trying to understand how a computer works as a whole. The book [1] plays a role in this by briefly presenting each of the topics and by providing insight into what goes on in the computer through various tasks, ranging from translating a high-level programming language into machine code to designing the hardware on which it can run.

Keywords: computer, hardware, software, operating system, central processing unit, compiler, virtual machine.

Poglavje 1

Uvod

Sodobni računalniki postajajo vse kompleksnejši, kar vodi do vedno težje vidne interakcije med strojno opremo, programsko opremo, prevajalniki in operacijskim sistemom. Delovanje računalnika je zato postalo težko razumljivo in skrito pod mnogimi sloji zapletenih vmesnikov. Ta kompleksnost se pozna tudi pri poučevanju računalništva, ki je postalo razdeljeno na mnogo področij, kjer vsako izmed njih pokriva le svoj del. To pa velikokrat vodi do specializiranosti na določenih področjih brez poznavanja celotnega delovanja računalnika.

Knjiga The Elements of Computing Systems [1] tako opisuje izdelavo preprostega računalnika in omogoča prikaz njegovega celotnega delovanja. Začne s predpostavko, da imamo podana vrata NAND ter pomnilno celico D, iz katerih nato zgradimo bolj kompleksna vezja, potrebna za izdelavo strojne opreme računalnika. Nato se nadaljuje z izgradnjo zbirnega jezika, navideznega stroja in visokonivojskega programskega jezika Jack, ki teče na prav tako implementiranem operacijskem sistemu Jack.

Izdelava računalnika, imenovanega Hack, izdelanega po navodilih iz knjige [1], na grobo poteka v dveh delih. Najprej se izdela strojni del, ki predstavlja osnovo za kasneje izdelano programsko opremo. V knjigi [1] avtorja dobro razložita vlogo in delovanje posamezne računalniške komponente, ki sestavlja računalnik, ter pripravita naloge z navodili za izdelavo posamezne kompo-

nente. Poglavja v knjigi so zastavljena tako, da se računalnik v vsakem poglavju nadgradi. V vsakem naslednjem se tako uporablja predhodno izdelane komponente, katere morajo biti dobro testirane za pravilno celotno delovanje računalnika.

Končni izdelek ni implementiran v fizični obliki na pravem vezju, ampak se zaganja v emulatorju CPE (centralno procesna enota), ki je program, implementiran s strani avtorjev, ki omogoča izvajanje strojne kode. Prav tako pa se bi računalnik dalo implementirati tudi v fizični obliki na FPGA (ang. "field-programmable gate array") vezju. Računalnik poleg glavnega pomnilnika za operande vsebuje še pomnilnik za ukaze, v katerega se naloži strojno kodo operacijskega sistema in programa, katerega hočemo poganjati. Kadar hočemo zamenjati program, ki se izvaja, pa je potrebno računalnik izključiti ter vanj ponovno naložiti strojno kodo želenega programa. Operacijski sistem Jack nima grafičnega uporabniškega vmesnika, ampak vsebuje le nekaj minimalističnih knjižnic, ki razširjajo programski jezik Jack ter jih prav tako vsebuje tudi večina sodobnih operacijskih sistemov. V sam računalnik je možno priklopiti le dve vhodno-izhodni napravi: zaslon in tipkovnico.

Diplomsko delo je razdeljeno na štiri poglavja. Najprej so predstavljeni uporabljeni programi in tehnologije. Programe, s katerimi smo si pomagali pri izdelavi računalnika, sta avtorja izdelala sama in tako omogočila lažji ter hitrejši razvoj. S tem smo se izognili učenju uporabe industrijskih tehnologij in programov, ki nam bi vzeli veliko več časa. Sledi izdelava strojne opreme, ki je podlaga za programsko opremo, izdelano v tretjem poglavju. V zadnjem delu pa je opis in prikaz razširitve računalnika, kjer smo implementirali množenje in deljenje na nivoju strojne opreme. Ti dve operaciji sta sicer v knjigi implementirani na plasti operacijskega sistema, kar ju naredi lažji za implementacijo, a nekoliko počasnejši pri izvajanju.

Poglavje 2

Uporabljene tehnologije in orodja

V tem poglavju so opisani programi in računalniški jeziki, ki se uporabljajo v procesu izdelave računalnika. Uporabljene so poenostavljene oblike tehnologij, katere se uporabljajo v industrijski izdelavi sodobnega računalnika.

2.1 HDL

HDL (ang. "hardware description language") je jezik za opis digitalnih vezij. HDL, uporabljen v knjigi [1], je podoben industrijskim HDL, kot sta Verilog in VHDL, vendar je preprostejši, saj se ga da usvojiti v eni uri. Avtorja knjige [1] sta se ravno zaradi preprostosti odločila za svoj HDL, ki je še vedno dovolj zmogljiv, da z njim opišemo vezje, potrebno za delovanje računalnika Hack ali katerega koli drugega računalnika, narejenega po načelih, opisanih v knjigi [1].

HDL se piše v tekstovni dokument s končnico `.hdl` in ima strukturo, kot jo prikazuje Slika 2.1. V dokumentu je najprej opis komponente (pod imenom "čip"), temu sledi njeno ime, imena vhodnih in izhodnih spremenljivk ter na koncu še implementacija.

Not.hdl

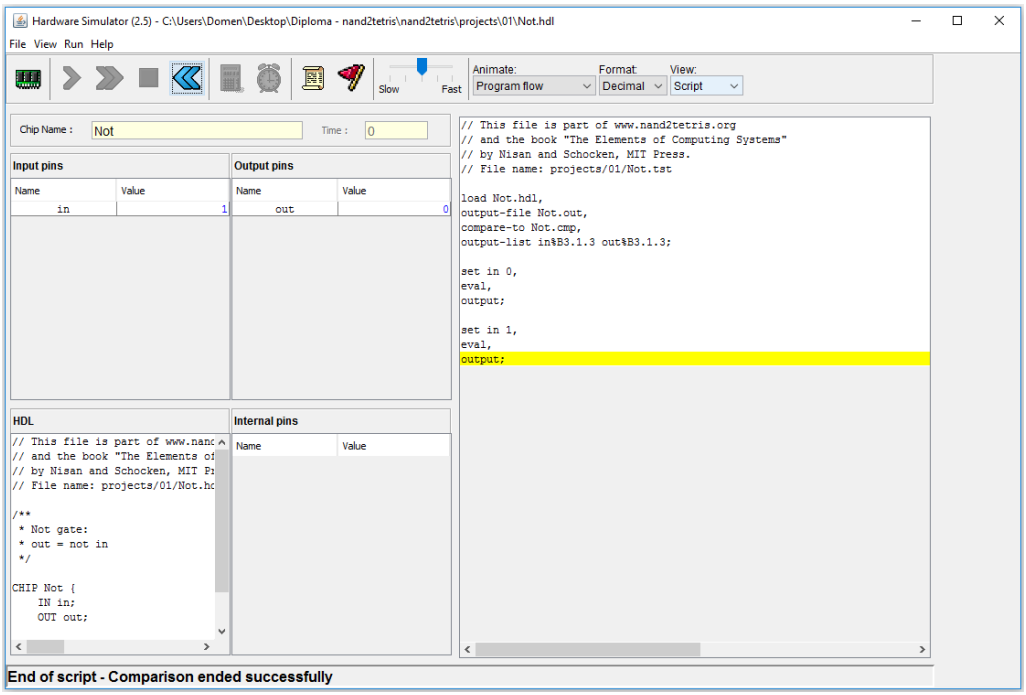
```
1  // This file is part of www.nand2tetris.org
2  // and the book "The Elements of Computing Systems"
3  // by Nisan and Schocken, MIT Press.
4  // File name: projects/01/Not.hdl
5
6  /**
7   * Not gate:
8   * out = not in
9   */
10
11 CHIP Not {
12     IN in;
13     OUT out;
14
15     PARTS:
16     Nand (a=in, b=in, out=out);
17 }
```

Slika 2.1: Primer HDL dokumenta.

2.2 Simulator strojne opreme

Simulator strojne opreme je program, ki se uporablja za simulacijo delovanja komponent še pred njihovo fizično implementacijo. Z njim se simulirajo komponente, katerih strukturni opis se nahaja v `.hdl` datotekah. Simulacija poteka tako, da za podane vhodne spremenljivke simulator izračuna in vrne izhodne in vmesne spremenljivke. Za preverjanje pravilnosti delovanja komponent se v simulator naloži testna skripta, katera vsebuje testne primere za delovanje posamezne komponente. Nato se glede na podane vhodne spremenljivke iz testnih primerov primerja izhode komponente z vrednostmi v primerjalni datoteki, ki vsebuje pravilne izhode za podane vhode.

Uporabniški vmesnik simulatorja vidimo na Sliki 2.2. V njem je prikazana koda komponente, polje za vnos vhodnih spremenljivk, prikaz izhodnih in vmesnih spremenljivk ter testna koda, katero se lahko izvršuje vrstico po vrstico ter sproti preverja delovanje komponente. Kode trenutne komponente se v simulatorju ne da neposredno spreminjati in je treba uporabiti zunanji urejevalnik.



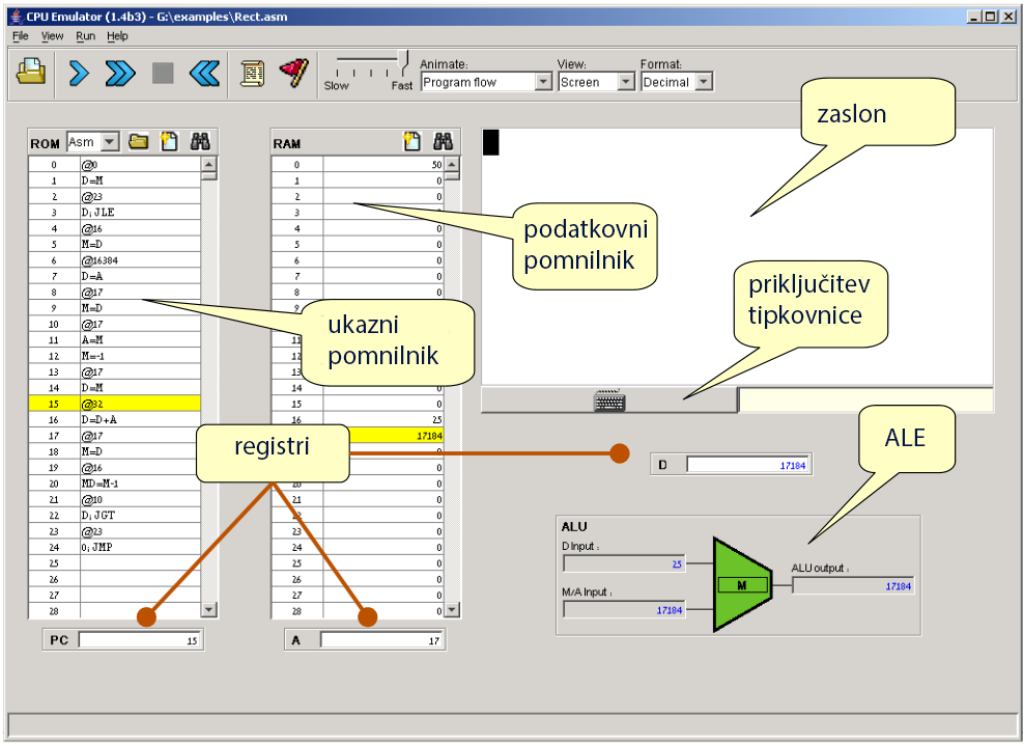
Slika 2.2: Uporabniški vmesnik simulatorja strojne opreme.

2.3 Emulator CPE

Emulator CPE se uporablja za testiranje programov še pred samo implementacijo CPE. Uporabniški vmesnik emulatorja se nahaja na Sliki 2.3. Za prikazovanje vsebine ima zaslon ločljivosti 512 x 256 pik. Vanj se naloži koda, napisana v zbirnem jeziku, ki je nato prikazana v ukaznem pomnilniku. Koda se lahko izvršuje brez ali pa z animacijo, pri kateri se pomikamo skozi vrstice programa in spremljamo trenutne vrednosti v registrih ter podatkovnem pomnilniku.

2.4 Zbirnik

Za prevajanje zbirne kode v strojno kodo smo izdelali program imenovan zbirnik. Da smo lahko preverili ali naša implementacija zbirnika deluje pravilno smo si pomagali s programom na Sliki 2.4, ki sta ga izdelala avtorja

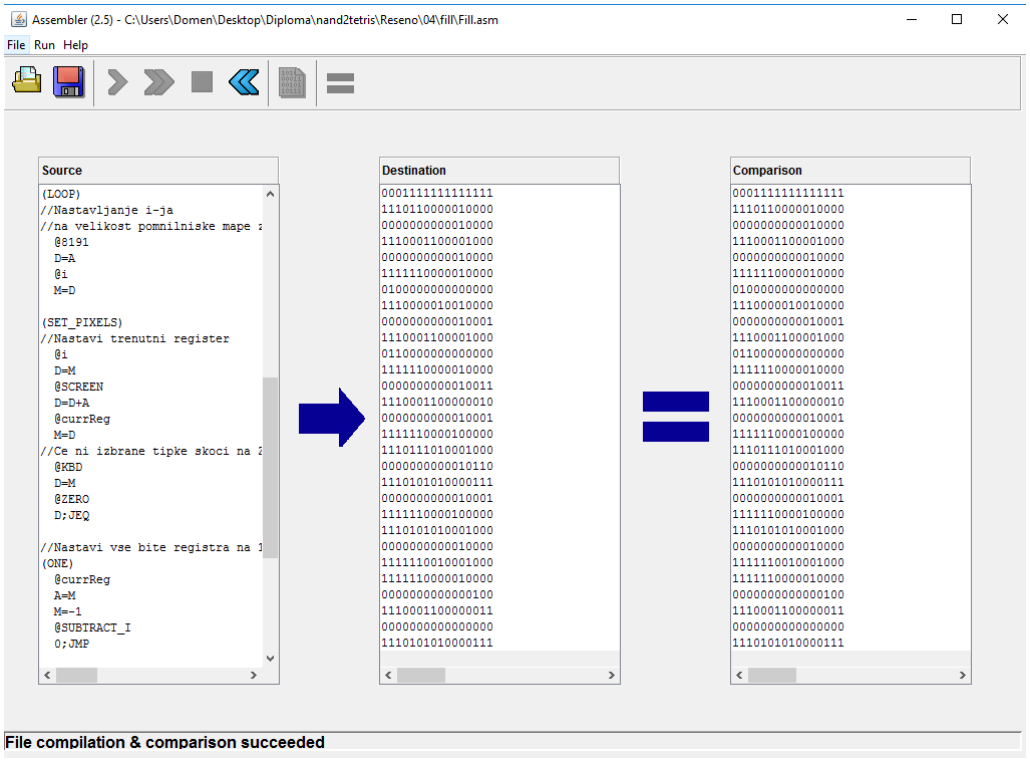


Slika 2.3: Uporabniški vmesnik emulatorja CPE.

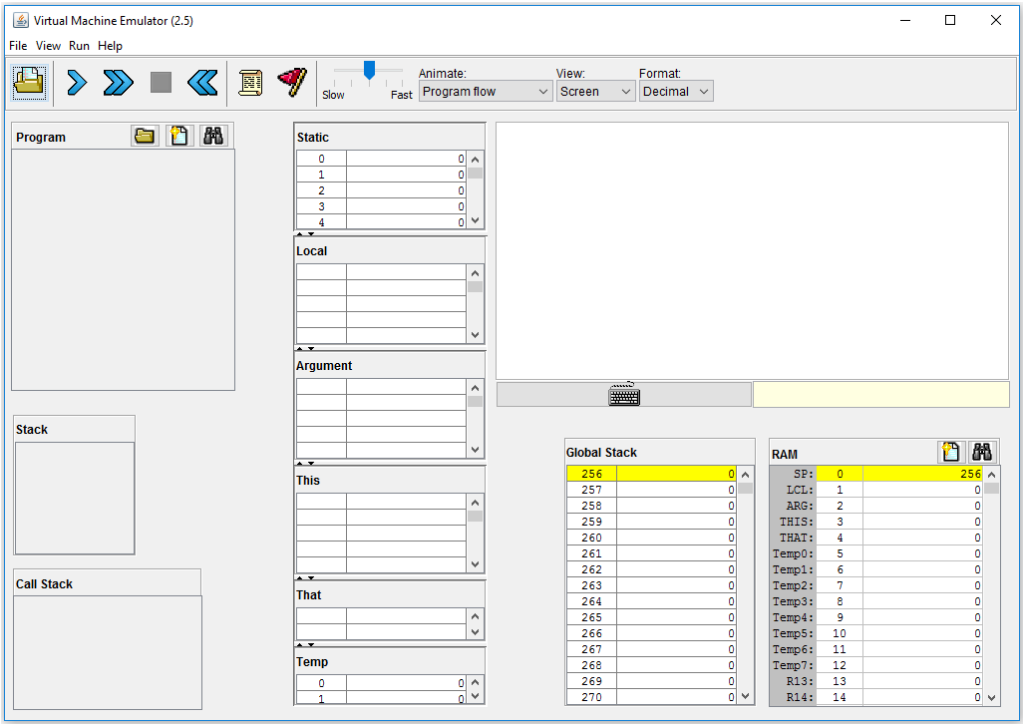
knjige [1]. Deluje tako da se vanj naloži zbirna in strojna koda, za katero nam pove ali naša implementacija zbirnika deluje pravilno.

2.5 Emulator navideznega stroja

Emulator navideznega stroja se uporablja za izvajanje prevedene kode iz višjenivojskega jezika Jack. Omogoča enostaven pregled nad segmenti v glavnem pomnilniku med samim izvajanjem programa in ima za lažje razumevanje z oznakami iz zbirnega jezika poimenovanih prvih šestnajst registrov. Njegov uporabniški vmesnik je prikazan na Sliki 2.5. Med izvajanjem omogoča tudi hierarhični prikaz vseh trenutno aktivnih funkcij.



Slika 2.4: Uporabniški vmesnik programa, namenjenega testiranju lastne implementacije zbirnika.



Slika 2.5: Uporabniški vmesnik emulatorja navideznega stroja.

Poglavje 3

Razvoj strojne opreme

V tem poglavju je opisana izdelava strojne opreme, za katero se nato v naslednjem poglavju izdela programska oprema. Začne se z izgradnjo osnovnih logičnih vrat in konča z izdelavo centralne procesne enote, na kateri je možno izvajati kodo strojnega jezik Hack.

3.1 Boolove funkcije in logična vrata

Razvoj računalnika Hack se začne s predpostavko, da imamo podana vrata NAND, iz katerih se nato zgradi vse ostale, bolj kompleksne komponente. Avtorja knjige [1] sta se odločila za izvedbo z vrati NAND, ker je vezje za izgradnjo le-teh bazično (pri implementaciji s tranzistorji), kar bi prišlo v poštev pri fizični implementaciji strojne opreme.

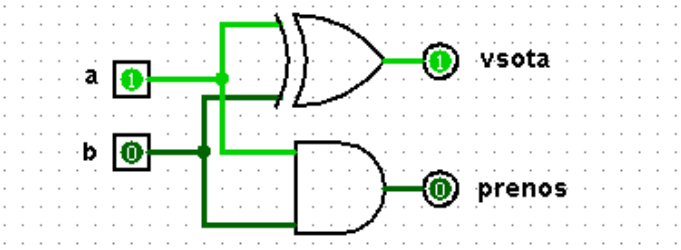
Računalnik Hack poleg osnovnih logičnih operatorjev vsebuje nekaj njihovih standardnih nadgradenj, prikazanih v Tabeli 3.1, ter ostale kokmponente, ki so razložene v nadaljevanju. Izvedbo posameznega logičnega operatorja se implementira na podlagi podane pravilnostne tabele. Iz nje se naredi disjunktivna normalna oblika, katero je potrebno z uporabo Boolove algebre kar se da poenostaviti (minimizirati) zaradi hitrosti izvajanja v računalniku. Pri tem si lahko pomagamo s Karnaughovim diagramom, iz katerega se da dokaj enostavno odčitati poenostavljen Boolov izraz. Iskanje najkrajšega izraza pri

Tabela 3.1: Logični operatorji.

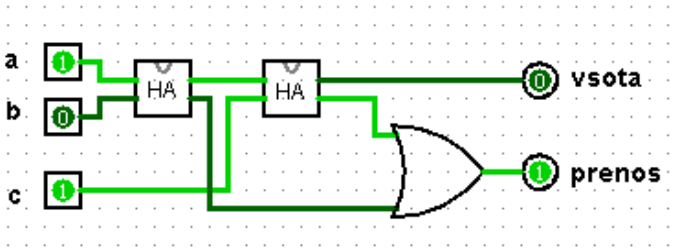
Logični operator	**Opis**
NE-IN	Primitivni operator. Izhod je 1, če je vsaj eden od vhodov 0.
NE	Izhod je negirana vrednost vhoda.
IN	Izhod je 1, če sta oba vhoda 1.
ALI	Izhod je 1, če je vsaj eden od vhodov 1.
izključujoči ALI	Izhod je 1, če je natanko eden izmed vhodov 1.
MUX	Če je signal 0, prenesi vhod 0 na izhod, drugače vhod 1.
DEMUX	Če je signal 0, prenesi vhod na izhod 0, drugače na izhod 1.
Ne16	16-bitni NE
In16	16-bitni IN
Ali16	16-bitni ALI
Mux16	16-bitni multipleksor
Ali8/1	ALI z osmimi vhodi
Mux4/1_16	Multipleksor s 4 16-bitnimi vhodi
Mux8/1_16	Multipleksor z 8 16-bitnimi vhodi
DMux4/1	Demultipleksor s 4 izhodi
DMux8/1	Demultipleksor z 8 izhodi

večjih komponentah, ki vsebujejo na stotine elementov, povezav in vhodov, je NP-težak problem, zato je potrebno kompleksne komponente razbiti na manjše dele, katere je lažje zgraditi in optimizirati.

Pravilnostne tabele za posamezen logični operator sta pripravila avtorja knjige [1], ki sta bila v vlogi sistemskega arhitekta. Njegova naloga pri izgradnji strojne opreme je, da večje komponente razdeli na več manjših in za vsako izdela vmesnik. Ta vsebuje ime komponente, imena vhodnih in izhodnih spremenljivk, testno skripto in primerjalno datoteko, v kateri so zapisani pravilni izhodi za določene vhode. Do njih pride tako, da logiko za določeno komponento implementira v enem izmed višjenivojskih jezikov. To omogoča testiranje strojne opreme, še preden se začne s pisanjem v HDL. Ko ima sistemski arhitekt vse pripravljeno in preizkušeno, začne razvijalec z



Slika 3.1: Vezje polovičnega seštevalnika (HA).



Slika 3.2: Vezje polnega seštevalnika.

izvedbo v HDL.

3.2 Boolova algebra in ALE

Aritmetično-logična enota izvršuje aritmetične, logične in primerjalne operacije nad podatki, ki jih zahtevajo ukazi. Za izvajanje teh operacij v računalniku Hack je potrebno poleg že obstoječih logičnih operatorjev implementirati še 16-bitni seštevalnik. Sestavlja ga šestnajst zaporedno vezanih polnih seštevalnikov, kjer je izhodni bit za prenos enega polnega seštevalnika vezan na vhod njegovega naslednika. Posamezen polni seštevalnik je sestavljen iz dveh polovičnih seštevalnikov v kombinaciji z logičnim operatorjem `ALI`, kot prikazuje Slika 3.2. Polovični seštevalnik pa je implementiran z logičnima operatorjema `izključojoči ALI` ter `IN`, kot je razvidno iz Slike 3.1. Ta na vhod dobi dva bita, za katera na izhodu vrne njuno vsoto ter prenos. Pri seštevanju več-bitnih števil lahko prva bita seštejemo

s polovičnim seštevalnikom, za vse nadaljnje pa potrebujemo polnega, saj se na vhodu poleg dveh bitov, ki ju seštevamo, pojavi tudi prenos predhodno seštetih bitov. Polni seštevalnik za razliko od polovičnega na vhod dobi tri bite, za katere vrne njihovo vsoto in prenos.

3.2.1 Negativna števila

Seštevalnik, uporabljen za pozitivna števila, lahko deluje tudi nad negativnimi. Če hočemo odšteti $y - x$ s seštevanjem, je potrebno negiran x prišteti y (3.1). Negacija binarnih števil, kjer so števila predstavljena z dvojiškim komplementom, tako poteka po formuli (3.2), kjer n predstavlja največje število bitov za prikaz števila. Kot je prikazano v formuli (3.2), število najprej pretvorimo v eniški komplement kateremu nato prištejemo ena. Kadar pri seštevanju pride do prenosa, ga seštevalnik ignorira, kar omogoča pravilen rezultat operacij v dvojiškem komplementu.

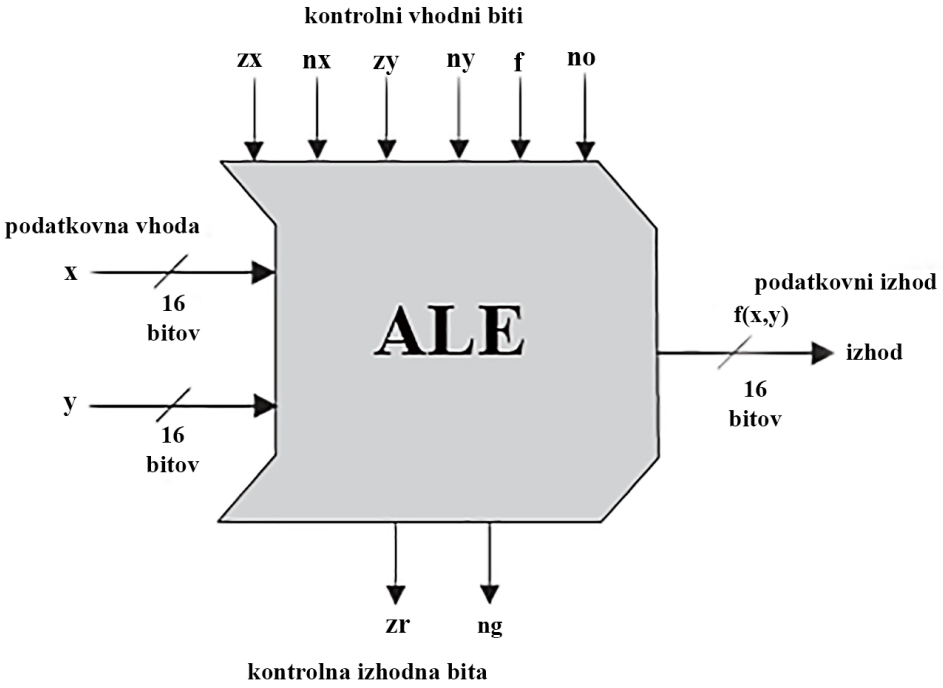
$$y - x = y + (-x) \tag{3.1}$$

$$-x = (2^n - x) = (2^n - 1) - x + 1 \tag{3.2}$$

Primer negacije 4-bitnega števila 4 je prikazan na Sliki 3.3. Število ima binarno vrednost 0100, kar odštejemo največjemu možnemu številu, ki ga je mogoče zapisati s 4 biti, nato pa prištejemuo število 0001 ter dobimo število 1100, kar v dvojiškem komplementu predstavlja -4.

```
  1111
-
  0100
  1011
+    1
  1100
```

Slika 3.3: Primer negacije števila 4.



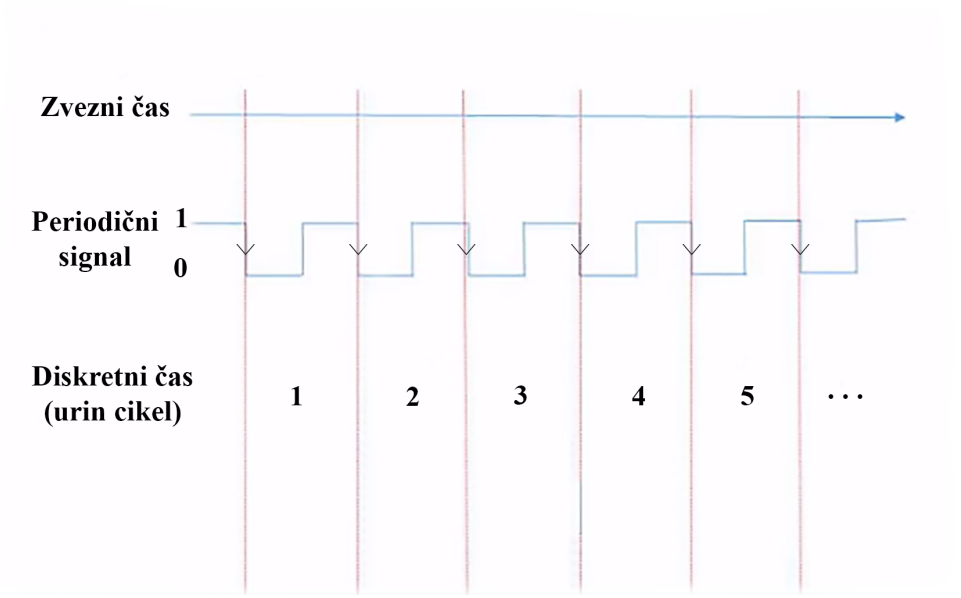
Slika 3.4: ALE Hack.

3.2.2 Aritmetično-logična enota Hack

Aritmetično-logično enoto (ALE), katere diagram je na Sliki 3.4, sestavljata dva 16-bitna vhoda za podatke, 16-bitni izhod za podatke, 2 kontrolna izhodna bita ter 6 kontrolnih vhodnih bitov, ki določajo, katera funkcija se naj izvede nad vhodnimi podatki. Število možnih funkcij, ki bi jih lahko Hack ALE glede na 6 kontrolnih bitov izvedel, je 64, saj je to največje možno število kombinacij enic in ničel med njimi. Ker je cilj narediti čim preprostejšo ALE, ima ta prednastavljenih le 18 osnovnih funkcij, ki se nahajajo v Tabeli 3.2, kjer je tudi razložena vloga posameznega kontrolnega bita. Med funkcijami ni množenja in deljenja, ki sta implementirana programsko na plasti operacijskega sistema. To naredi operaciji nekoliko počasnejši, vendar omogoča preprostejšo implementacijo aritmetično logične enote.

Tabela 3.2: Pravilnostna tabela ALE.

zx	**nx**	**zy**	**ny**	**f**	**no**	**izhod**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

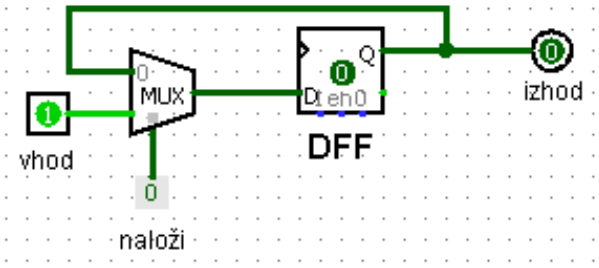


Slika 3.5: Predstavitev časa v računalniku. V prikazanem primeru imamo proženje na zadnjo fronto ure.

3.3 Sekvenčna vezja

Izhodi v kombinacijskih vezjih so odvisni le od kombinacij podatkov na vhodu in ne morejo vzdrževati njihovega stanja. Za pravilno delovanje računalnika mora ta imeti zmožnost pomnjenja. Podatki se zato shranjujejo v pomnilne celice. Sekvenčna vezja so sestavljena iz kombinacijskih vezij in pomnilnih celic, katerih primeri so registri in števci.

V sekvenčnem vezju so izhodi odvisni od trenutnih vhodov in od stanja, zato mora vezje za svoje delovanje upoštevati čas. Tako se lahko izbrane operacije izvajajo v določenem časovnem ciklu ter posledično omogočajo sinhronizirano delovanje vezja. V računalniku se za ta namen nahaja ura, katera na vhode sekvenčnih vezij oddaja periodični signal, predstavljen s ponavljajočim se zaporedjem ničel in enic. Čas je predstavljen diskretno, kot prikazuje Slika 3.5. V vsakem urinem ciklu se v posamezni komponenti izvede le en ukaz.



Slika 3.6: Diagram pomnilne celice (shranjevanje enega bita).

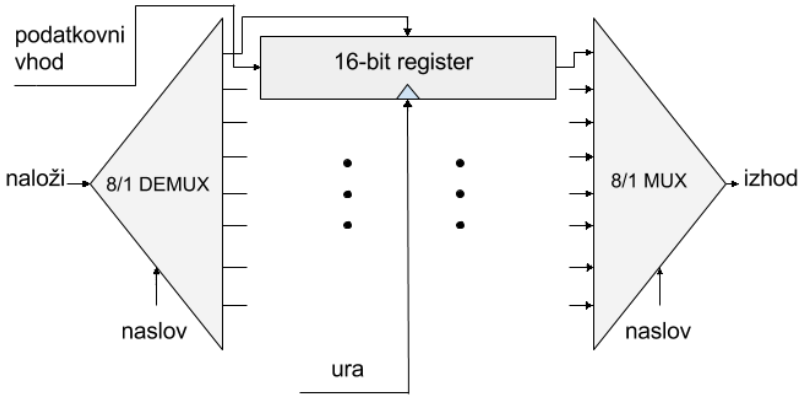
3.3.1 Pomnilnik

Pomnilni element, ki hrani en bit informacije v računalniku Hack, je `pomnilna celica D`. Zapomni si vhod iz prejšnjega urinega cikla in ga prikaže na izhodu v naslednjem ciklu. Ena izmed implementacij `pomnilne celice D` uporablja kombinacijo vrat NAND. V knjigi [1] sta se avtorja odločila za uporabo DFF (ang. "D flip-flop") kot primitivne komponente, ker sta hotela ločiti način uporabe kombinacijskega vezja z načinom uporabe sekvenčnega vezja. `Pomnilna celica D` vsebuje zanko, ki se je v kombinacijskem vezju naj ne bi uporabljalo. Simulator strojne opreme, katerega sta razvila avtorja, tako omogoča zanke le s komponentami, v katerih se pojavlja `pomnilna celica D` in ob zanki s kombinacijskimi komponentami javi napako, torej ne podpira asinhronskih sekvenčnih vezij. Le-ta so potencialno hitrejša od sinhronskih vendar pa je njihovo načrtovanje zapleteno.

DFF lahko shrani en bit informacije le za en urin cikel, zato je potrebno implementirati register, ki lahko hrani bit informacije dalj časa. Diagram takega registra prikazuje Slika 3.6. Poleg vhodov za podatke se nahaja še kontrolni bit, ki pove, kdaj se naj shrani nova vrednost.

Računalnik Hack ima 16-bitni procesor, zato so potrebni tudi registri te velikosti. Dobimo jih tako, da zaporedno povežemo 16 pomnilnih celic. Iz njih nato zgradimo glavni pomnilnik (RAM) poljubne velikosti[1]. Vsak register v RAM (ang. "random access memory") ima svoj naslov, preko

[1]Pri tem računalniku je RAM sestavljen kar iz registrov, kar v realnem svetu ni običaj.

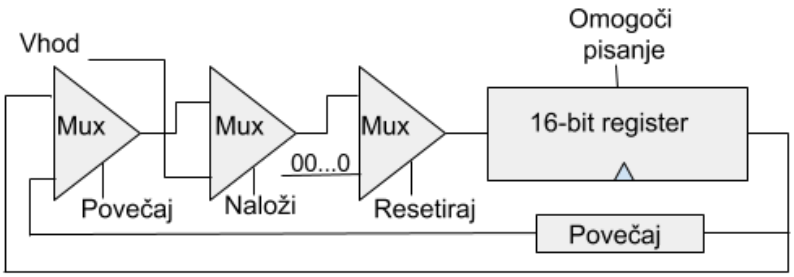


Slika 3.7: RAM vezje.

katerega se dostopa do njega. Potrebno število bitov v naslovu je $\log_2 n$, kjer je n število pomnilnih celic v RAM. V danem trenutku lahko dostopamo le do enega registra, nad katerim izvajamo operacijo branja ali pisanja. Dobra lastnost RAM je ta, da je dostopni čas do vseh registrov enak. Vezje za izgradnjo tega je prikazano na Sliki 3.7, kjer je primer 128-bitnega RAM.

3.3.2 Števec

Števci so sekvenčne komponente, ki igrajo pomembno vlogo v digitalnih vezjih. Tipična CPE uporablja programski števec, katerega vsebina pove naslov ukaza, ki ga je potrebno izvesti kot naslednjega. Kot pove že ime, je števec namenjen povečevanju nekega števila za določeno konstanto. V našem primeru se v vsaki iteraciji število poveča za ena. Zraven mora imeti implementirano še ponastavljanje števila na privzeto vrednost in možnost spremembe na poljubno novo število. Diagram programskega števca, uporabljenega v računalniku Hack, je prikazan na Sliki 3.8.



Slika 3.8: Diagram števca.

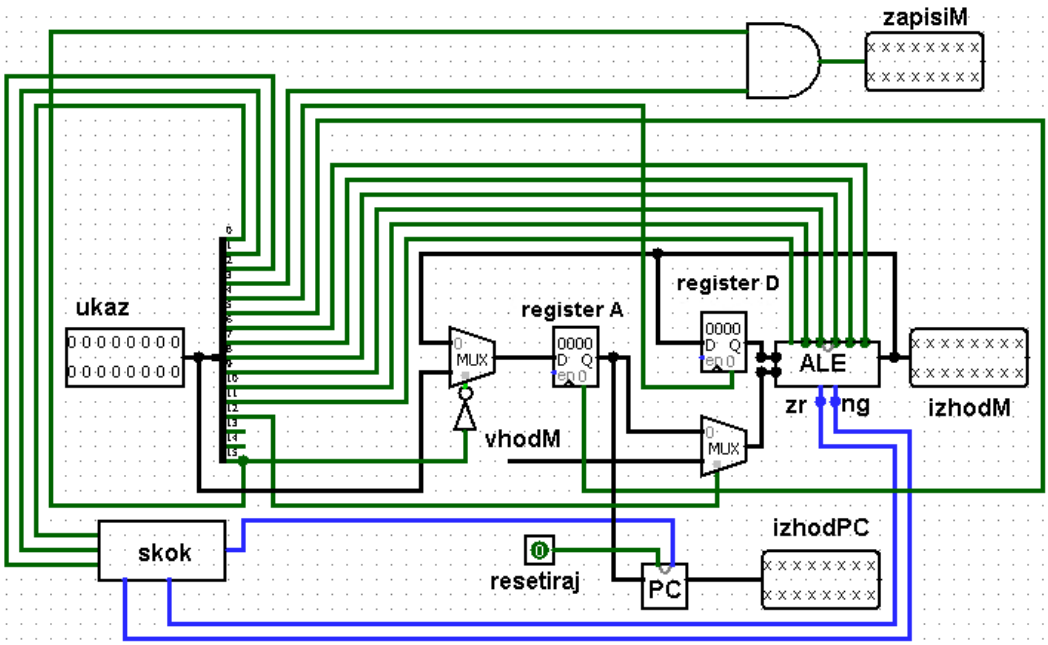
3.4 Strojni jezik

Strojni jezik je nizkonivojski programski jezik, ki povezuje strojno in programsko opremo. Zapisan je v binarni obliki (digitalni električni signali: visok in nizek), da je razumljiv CPE. Sestavljajo ga nizi končne dolžine, ki predstavljajo ukaze in operande, nad katerimi se ti ukazi izvršujejo. Strojni jezik je odvisen od arhitekture centralne procesne enote, katera se med procesorji razlikuje ter tako posledično onemogoča prenosljivost jezika. Zapis je človeku težko razumljiv, zato se programe piše v zbirnem jeziku, ki uporablja za zapis mnemonike, kateri so nato s pomočjo zbirnika prevedeni v strojno kodo, ki jo razume računalnik.

3.4.1 Pomnilniška hierarhija

Računalnik Hack ima pomnilnik razdeljen na dva dela: eden je namenjen ukazom in drugi operandom. Registri v pomnilniku so velikosti 16 bitov, za naslove pa je namenjenih 15 bitov. To omogoča pomnilnik velikosti 2^{15} = 32K 16-bitnih registrov.

Branje podatkov iz glavnega pomnilnika je za centralno procesno enoto počasna operacija, katero lahko pohitrimo z uporabo pomnilniške hierarhije. Z njo želimo doseči, da bi bil velik in poceni pomnilnik videti kot manjši, izdelan v bistveno hitrejši tehnologiji. Za doseganje tega se na poti do CPE postavi več pomnilnikov, kateri so z bližanjem CPE vedno manjši in izde-



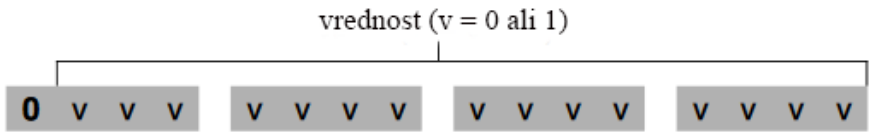
Slika 3.9: Diagram CPE.

lani s tehnologijo, ki omogoča hitrejše prenose. Računalnik Hack v ta namen poleg glavnega pomnilnika uporablja dva registra, poimenovana A (ang. "address") in D (ang. "data"), katera sta del CPE. Oba sta namenjena hranjenju podatkov. Glede na izbran ukaz, ki se bo izvršil, pa je A lahko namenjen še naslavljanju registrov v glavnem pomnilniku.

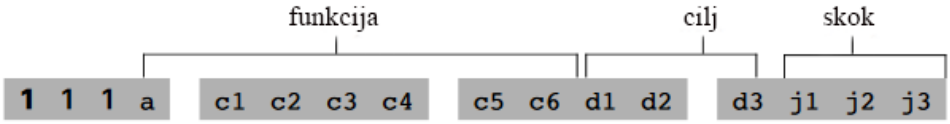
V splošnem pa imajo današnji računalniki med CPE in glavnim pomnilnikom dva ali tri nivoje predpomnilnikov. Pomnilniško hierarhijo pa sestavlja še pomnilnik, namenjen trajnemu hranjenju podatkov.

3.4.2 Ukazi strojnega jezika Hack

Strojni jezik računalnika Hack za vso delo, ki ga opravlja, potrebuje le dva ukaza, poimenovana A (ang. "address instruction") in C (ang. "compute instruction"), ki sta dolga 16 bitov. Za lažjo predstavitev njune uporabe je na Sliki 3.9 prikazan diagram celotne CPE. Najbolj pomemben bit ukaza se uporablja za določanje vrste ukaza, ničla pomeni, da gre za A, enica pa za



Slika 3.10: Ukaz A.



Slika 3.11: Ukaz C.

ukaz C.

Format ukaza A je predstavljen na Sliki 3.10. Je preprost ukaz, ki se ga uporablja le za shranjevanje 15-bitnih pozitivnih števil v register A. Najbolj pomemben bit določa vrsto ukaza in zavzame bit za predstavitev negativnih števil. Ta so tako dostopna le iz glavnega pomnilnika. Vrednost ukaza A je lahko tudi naslov registra v pomnilniku za ukaze, kjer se nahaja naslednji ukaz za izvršitev, ali pa naslov v podatkovnem pomnilniku, kjer se nahaja operand, kateri bo uporabljen v naslednjem ukazu. Sintaksa, ki se uporablja v zbirnem jeziku, je `@vrednost`. Ker se ukaz A lahko uporablja za naslavljanje v podatkovnem pomnilniku, ta vedno na izhod pomnilnika vrne vrednost registra na tem naslovu. V zbirnem jeziku se na to vrednost lahko sklicujemo s črko M (ang. "memory").

Nekoliko bolj kompleksen ukaz, ki upravlja večino dela, pa je ukaz C. Njegova struktura je prikazana na Sliki 3.11. Za najbolj pomembnim bitom sta dva bita, ki se ne uporabljata in sta nastavljena na ena. Nato sledi 7 bitov, s katerimi se določa operacijo, katero bo ALE izvedla nad vhodnimi podatki. Vse možne operacije so prikazane v Tabeli 3.2 in se jih izbira glede na stanje bitov, ki so predstavljeni s črko `c`. Bit, ki je označen s črko `a`, pa določa, ali se bo vrednost registra A uporabila kot takojšnja vrednost

Tabela 3.3: Registri, v katere se shrani rezultat iz ALE, glede na stanje bitov `d`.

d1	d2	d3	Mnemonik
0	0	0	null
0	0	1	M
0	1	0	D
0	1	1	MD
1	0	0	A
1	0	1	AM
1	1	0	AD
1	1	1	AMD

(konstanta) ali kot naslov registra v glavnem pomnilniku. Biti, označeni s črko `d`, določajo, v katere registre se bo shranil rezultat iz ALE. Možnih je 8 različnih kombinacij, prikazanih v Tabeli 3.3. Zadnji trije biti, označeni z črko `j`, pa določajo ukaz, ki se bo izvedel kot naslednji. Odločitev je odvisna od izhoda aritmetično-logične enote. Ta se primerja s številom 0 in če je kateri od pogojev v Tabeli 3.4 resničen, se zgodi skok na ukaz, shranjen v ROM na naslovu, ki je shranjen v registru A. Če je vrednost `j` bitov `null` ali ni izpolnjen noben pogoj, se izvrši naslednji ukaz v vrsti. Sintaksa, uporabljena v zbirnem jeziku, je `cilj=funkcija;skok`, kjer se lahko podatka za `cilj`, ki določa register za shranjevanje, in `skok`, izpustita.

3.4.3 Vhodno-izhodne naprave

Računalnik Hack ima možnost priključitve zaslona in tipkovnice, ki skrbita za komunikacijo z njim. Ta poteka s pomočjo pomnilniške preslikave, ki se nahaja v RAM in vsebuje trenutno stanje priključene naprave.

Zaslon ima 256 vrstic, v vsaki vrstici pa je 512 pik. Pomnilniška preslikava za zaslon se začne v RAM na naslovu 16384 in zasede 8 K registrov. Posamezna pika je v pomnilniku predstavljena kot bit, ki sporoča piki, naj

Tabela 3.4: Pravilnostna tabela možnih skokov v C ukazu.

j1	j2	j3	Mnemonik	Efekt
0	0	0	null	Ni skoka
0	0	1	JGT	Če `izhod` > 0 skok
0	1	0	JEQ	Če `izhod` $= 0$ skok
0	1	1	JGE	Če `izhod` ≥ 0 skok
1	0	0	JLT	Če `izhod` < 0 skok
1	0	1	JNE	Če `izhod` $\neq 0$ skok
1	1	0	JLE	Če `izhod` ≤ 0 skok
1	1	1	JMP	Skok

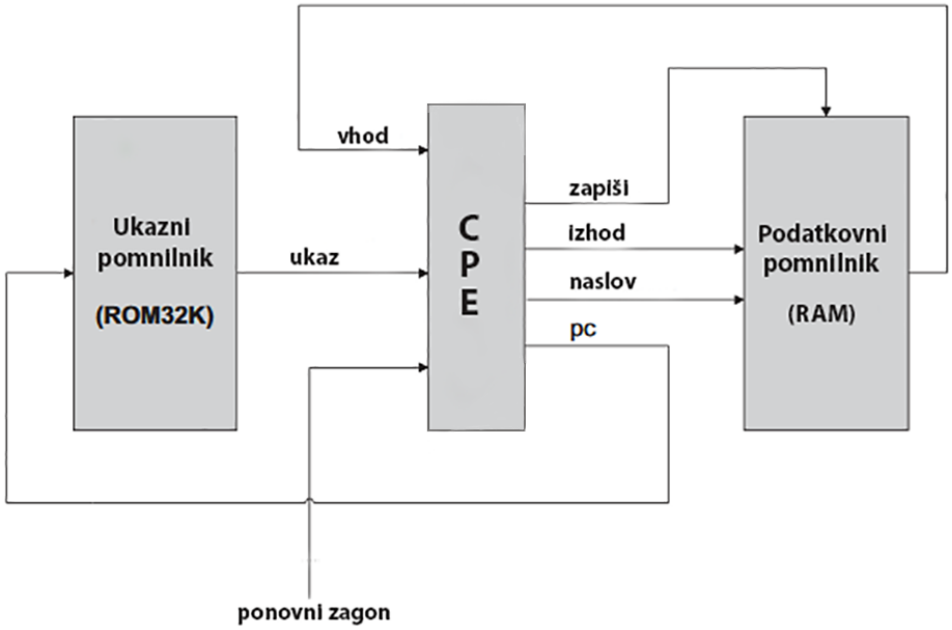
se obarva črno, če je nastavljen na ena, sicer naj bo bela. Tako register na naslovu 16384 določa obarvanost prvih 16-pik v zgornjem levem robu.

Zaznavanje trenutno izbrane tipke na tipkovnici pa poteka preko registra na naslovu 24576, v katerem je shranjena ASCII vrednost izbrane tipke. Kadar ni v uporabi nobena tipka, je vrednost nastavljena na 0.

3.5 Arhitektura računalnika

Arhitektura Hacka je bolj minimalna, saj tipični računalniki vsebujejo več registrov, zmogljivejše ALE in bogatejši nabor ukazov. Večini digitalnih računalnikov je podoben po tem, ker temelji na `von Neumannovemu` modelu. Spada med *namenske* računalnike, saj ima kodo za izvrševanje zapisano v ROM (ang. "read-only memory"), kjer se nahaja le en program in je potrebno za izvajanje drugega programa zamenjati ROM. Nanj je možno priključiti dve vhodno-izhodni napravi, zaslon, ki prikazuje le črno belo sliko, in tipkovnico. Velikost bralnega pomnilnika, v katerem so zapisani ukazi, je 32 K 16-bitnih registrov.

Računalnik Hack je sestavljen iz pomnilniške enote in CPE. Pomnilniška enota je razdeljena na pomnilnik za podatke RAM in bralni pomnilnik za ukaze ROM. CPE pa sestavljajo ALE, registri in kontrolna enota. Arhitek-



Slika 3.12: Arhitektura računalnika Hack.

tura je prikazana na Sliki 3.12.

3.5.1 Kontrolna enota

Kontrolna enota skrbi za dekodiranje ukazov pred izvajanjem in tako signalizira določenim komponentam v CPE, kako izvršiti ukaz. ALE tako dobi signal, katero operacijo izvršiti nad vhodnimi podatki ter iz katerih registrov prebrati podatke za računanje, registrom in pomnilniku za podatke pa je signalizirano, ali naj shranijo nove vrednosti. Kontrolna enota skrbi tudi za izbiro ukaza, ki se bo izvajal v naslednjem urinem ciklu. To je implementirano tako, da signali iz trenutnega ukaza glede na izhod ALE sporočijo programskemu števcu, ali se mora zgoditi skok v ukaznem pomnilniku, sicer se izvrši ukaz, zapisan v naslednjem registru v vrsti. Izhod programskega števca je povezan na vhod ukaznega pomnilnika ter tako določa naslov registra z ukazom. Ob pritisku gumba reset se programski števec postavi na vrednost 0 in

tako začne program v ROM izvajati od začetka. Izhod CPE je povezan na vhod pomnilnika za podatke in tako shranjuje na novo izračunane podatke na določen naslov v pomnilniku. Novi podatek se, če je tako signalizirano s strani ukaza, shrani tudi v registra znotraj CPE in je ob ponovni uporabi hitreje dostopen.

Delovanje CPE je tako ponavljajoča se zanka, v kateri se najprej prebere ukaz iz ukaznega pomnilnika, se ga dekodira, izvrši in določi naslednji ukaz za izvršitev. Možne naloge, ki se lahko zgodijo v CPE, so računanje nove vrednosti v ALE, spreminjanje vrednosti registrov v CPE, branje ali pisanje v pomnilnik ter določanje naslednjega ukaza.

Poglavje 4

Razvoj programske opreme

V tem poglavju je opisana izdelava programske opreme, ki se poganja na strojni opremi, izdelani v prejšnjem poglavju. Začne se z implementacijo zbirnika, kateri povezuje težko berljivo strojno kodo z zbirnim jezikom, nadaljuje pa z izgradnjo prevajalnika in osnovnega operacijskega sistema, kateri omogoča poganjanje objektno usmerjenega programskega jezika Jack.

4.1 Zbirnik

Preden se lahko nek program začne izvajati na računalniku, ga je potrebno prevesti v strojno kodo. Prevajanje se izvede s pomočjo programa, imenovanega zbirnik, ki prevede zapis mnemonično zapisanega ukaza. Ta na vhod dobi ukaze v zbirnem jeziku in na izhod vrne ukaze, zapisane v binarni obliki, kateri so nato naloženi v pomnilnik in razumljivi strojni opremi, ki jih izvaja. Za prevajanje je potrebno poznati celotno dokumentacijo zbirnega jezika in binarno preslikavo posameznega mnemonika. S temi specifikacijami strojnega jezika je načeloma možno prekodiranje opraviti ročno, vendar je to zelo počasen postopek, kjer je velika verjetnost pojavitve napak.

Postopek prevajanja iz zbirnega v strojni jezik je sestavljen iz naslednjih delov:

- Najprej je potrebno vsak ukaz razčleniti na manjše dele, iz katerih je sestavljen.
- Posamezen del je potrebno prevesti v njegov binarni zapis.
- Nadomestiti vse simbole z njihovimi naslovi v pomnilniku.
- Združiti binarne dele ukaza v celoto, ki tvori strojni ukaz.

4.1.1 Zbirni jezik Hack

Programi v zbirnem in strojnem jeziku so zapisani v tekstovnih datotekah s končnico `.asm` za zbirni in `.hack` za strojni jezik. Datoteko zbirnega jezika Hack sestavljajo tekstovne vrstice, kjer posamezna vrstica vsebuje enega izmed ukazov, opisanih v poglavju 3.4.2 ali pa je v njej deklariran simbol. Simboli so ena izmed glavnih funkcij zbirnika. Uporablja se jih za naslavljanje registrov v glavnem pomnilniku, katerim podamo ime, kar omogoča lažje in hitrejše pisanje ter berljivost kode, saj so lažje razumljivi kot pa naslovi registrov.

Hack vsebuje tri vrste simbolov:

- Primitivni simboli, ki so del zbirnega jezika, našteti v Tabeli 4.1, se uporabljajo večkrat in jih tako ni potrebno vedno znova deklarirati. Med njimi je tudi prvih 16 registrov, pred katerimi se ob naslavljanju na njih piše črka `R`. Ti registri so namenjeni vmesnemu shranjevanju in se nanje sklicujemo večkrat kot na ostale ter tako že ob prvem pogledu na kodo vemo, da gre za naslov registra in ne neko konstanto.
- Oznake, ki so psevdo-ukazi, saj se ne prevedejo v strojni jezik. Z njimi se sklicujemo na naslov ukaza, ki se nahaja prvi v vrsti za določeno oznako. To oznako nato uporabljamo v kodi, kadar hočemo narediti skok na ukaz, na katerega kaže oznaka. Sintaksa za deklariranje oznak je znotraj oklepajev "`(OZNAKA)`".

Tabela 4.1: Primitivni simboli, ki so del zbirnega jezika Hack.

Oznaka	RAM naslov	Opis
SP	0	kazalec na vrh sklada
LCL	1	kazalec na segment lokalnih spremenljivk
ARG	2	kazalec na segment argumentov
THIS	3	kazalec na segment trenutnega objekta
THAT	4	kazalec na segment trenutne tabele
R0-R15	0-15	sklicevanje na prvih 15 registrov
SCREEN	16384	začetek pomnilniške mape zaslona
KBD	24576	pomnilniška mapa tipkovnice

- Simboli, ki niso primitivni in ki niso deklarirani kot oznake, so spremenljivke. Ob vsaki deklaraciji nove spremenljivke se tej določi prvi prost register od naslova 15 naprej v podatkovnem pomnilniku.

Simboli so lahko sestavljeni iz zaporedja črk, številk, podčrtaja, pike, znaka za dolar in dvopičja, le začeti se ne smejo s številom. Komentarji se začnejo z dvema poševnicama (//) in končajo na koncu vrstice. Pri prevajanju v strojni jezik so ignorirani, prav tako se ignorirajo presledki in prazne vrstice. Vsi mnemoniki v zbirnem jeziku Hack morajo biti napisani z velikimi črkami, za oznake in spremenljivke pa velja, da so oznake napisane z velikimi, spremenljivke pa z malimi črkami.

4.1.2 Postopek prevajanja v strojno kodo

Za implementacijo zbirnika smo izbrali programski jezik Java, je pa možna implementacija v katerem koli višjenivojskem programskem jeziku. V zbirnem jeziku je možno sklicevanje na oznake še preden so te deklarirane, kar omogoča skoke v kodi. Da računalnik ve, v kateri del kode je potrebno skočiti, je potrebno zgraditi tabelo, v kateri je za vsako oznako zapisana vrstica kode, na katero kaže. Naša implementacija zbirnika tako dvakrat iterira skozi datoteko zbirnega jezika. V prvi iteraciji se zgradi le tabela z oznakami, ki se nato

```
1  0001111111111111
2  1110110000010000
3  0000000000010000
4  1110001100001000
5  0000000000010000
6  1111110000010000
7        .
8        .
9        .
```

Slika 4.1: Primer prvih nekaj vrstic programa v strojnem jeziku, prikazanega na Sliki 4.2.

uporablja v drugi iteraciji pri prevajanju kode. Tabela, v kateri so shranjene oznake, vsebuje še predefinirane simbole in spremenljivke, katere kažejo na registre v glavnem pomnilniku. Primitivni simboli so dodani v tabelo še pred prvo iteracijo, spremenljivke pa se dodajajo v tabelo v drugi iteraciji in se jim določi naslov prvega prostega registra od naslova 15 naprej. Še pred prvo iteracijo se zgradi tudi tabela mnemonikov, katera vsebuje njihove binarne preslikave. Ko se posamezen ukaz razčleni na manjše dele, se tako v tabeli za vsak del poišče njegova binarna preslikava in se sestavi strojni ukaz, sestavljen iz ničel in enic. V primeru, ko ukaz v zbirniku sestoji iz neke konstante in ne iz mnemonika, se to konstanto prevede v dvojiško število.

Hack zbirnik je zgrajen na predpostavki, da se vsak ukaz v zbirnem jeziku preslika v en ukaz strojnega jezika, kar je naiven pristop, saj se tipično v industrijskih zbirnih jezikih lahko en ukaz preslika v več ukazov strojnega jezika. Prav tako je naivna predpostavka ta, da je vsaka spremenljivka predstavljena z le enim registrom, kar omogoča največje število velikosti 16 bitov.

Primer zbirnega jezika Hack je prikazan na Sliki 4.2, kjer se nahaja koda, ki ob pritisku tipke na tipkovnici zaslon obarva v črno. Za lažje razumevanje je zraven podana tudi psevdo-koda. Na Sliki 4.1 pa je prikazana njegova preslikava v strojni jezik.

```
// while(true){
//   i = 8191; //velikost pomnilniske mape zaslona
//   while(0 < i){
//     currReg = SCREEN+i;
//     if(keyboard){
//       currReg = 1;
//     }else{
//       currReg = 0;
//     }
//     i--;
//   }
// }

(LOOP)
  //Nastavljanje i-ja
  //na velikost pomnilniske mape zaslona
  @8191
  D=A
  @i
  M=D

(SET_PIXELS)
  //Nastavi trenutni register
  @i
  D=M
  @SCREEN
  D=A+D
  @currReg
  M=D
  //Ce ni izbrane tipke skoci na ZERO
  @KBD
  D=M
  @ZERO
  D;JEQ

//Nastavi vse bite registra na 1
(ONE)
  @currReg
  A=M
  M=-1
  @SUBTRACT_I
  0;JMP

//Nastavi vse bite registra na 0
(ZERO)
  @currReg
  A=M
  M=0

(SUBTRACT_I)
  @i
  M=M-1
  D=M
  @SET_PIXELS
  D;JGE
  //Ce so spremenjeni vsi registri skok na LOOP
  @LOOP
  0;JMP
```

Slika 4.2: Primer programa v zbirnem jeziku, ki ob pritisku tipke zaslon obarva črno.

Tabela 4.2: Razdelitev glavnega pomnilnika

RAM naslov	**Uporaba**
0-15	Opis uporabe se nahaja v Tabeli 4.1
16-255	Statične spremenljivke
256-2047	Sklad
2048-16483	Shranjevanje objektov in tabel
16484-24575	Pomnilniška preslikava

4.2 Navidezni stroj

Prevajanje visokonivojskega programskega jezika Jack, ki se izvaja na računalniku Hack, poteka v dveh fazah. Najprej se koda prevede v vmesni jezik, nato pa jo navidezni stroj prevede v zbirni jezik. Prednost dvostopenjskega prevajanja je prenosljivost programov, napisanih v visokonivojskem jeziku med različnimi operacijskimi sistemi in procesorji, ki imajo implementiran navidezni stroj, ki zna prevesti vmesno kodo v ciljni nizkonivojski jezik.

Kot večina programskih jezikov tudi jezik navideznega stroja vsebuje aritmetične operacije, upravljanje pomnilnika, pogojne stavke in funkcije. Navidezni stroj, ki smo ga implementirali, shranjuje operande in rezultate operacij v podatkovno strukturo sklad. V njo vstavljamo in odstranjujemo elemente iz vrha sklada, kar omogoča implementacijo vseh programov ne glede na vrsto programskega jezika.

4.2.1 Sklad

V računalniku Hack se začetek sklada nahaja v glavnem pomnilniku na naslovu 256, kot prikazuje Tabela 4.2, v kateri je opisana uporaba segmentov glavnega pomnilnika. Za vstavljanje ali odstranjevanje elementov iz sklada je potrebno vedeti, kje se nahaja njegov vrh. Za to se uporablja kazalec, ki kaže na vrh sklada in se povečuje oziroma manjša, ko se element doda ali odvzame iz sklada. Za lažjo uporabo kazalca ima ta v zbirnem jeziku prede-

finiran simbol `SP` (angleško "stack pointer"), kateri kaže na naslednjo prosto lokacijo na vrhu sklada.

Za navidezni stroj je vsak program, ki ga izvaja, zbirka ene ali več funkcij. Zato se poleg kazalca na sklad uporabljata še kazalca, ki kažeta na segmenta, kjer se nahajajo lokalne spremenljivke in argumenti trenutne funkcije. Ta kazalca kažeta na začetek segmentov in je potrebno pri sklicevanju na njiju tudi podati, kateri operand v segmentu hočemo. Naslovi kazalcev in njihove oznake v zbirnem jeziku se nahajajo v Tabeli 4.1. Če hočemo na sklad vstaviti neko konstanto, to naredimo z virtualnim segmentom `constant`. Primer, v katerem bi vstavili število 4, bi bil `push constant 4`.

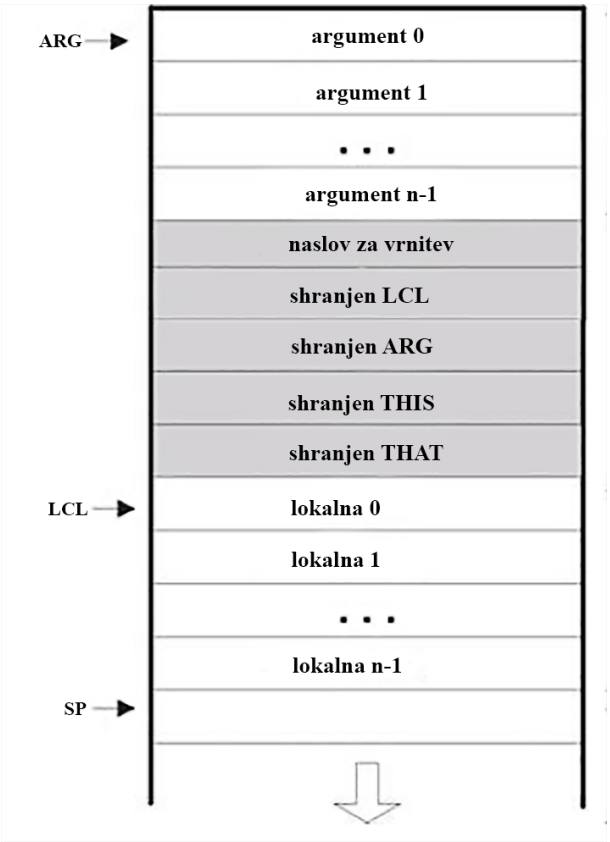
Pri klicanju nove funkcije klicoča funkcija vedno shrani svoje stanje na sklad, da se lahko ta po vrnitvi rezultata klicane funkcije obnovi. Primer klica je prikazan na Sliki 4.3. Najprej klicoča funkcija na sklad vstavi potrebne argumente, ki jih potrebuje klicana funkcija, nato na sklad shrani naslov, kamor je potrebno vrniti rezultat in ostale kazalce, ki so potrebni za obnovitev funkcije. Za tem vstavi še potrebno število lokalnih argumentov, čemur sledi prostor za sklad klicane funkcije.

Aritmetične operacije na skladu iz njega odstranijo potreben operand ali operanda ter rezultat nato vstavijo na vrh sklada, kot prikazuje Slika 4.4, na kateri je primer seštevanja. Na zelo podoben način deluje tudi Boolova algebra, katera kot rezultat vrne `true` (v dvojiškem komplementu število -1) in `false` (število 0).

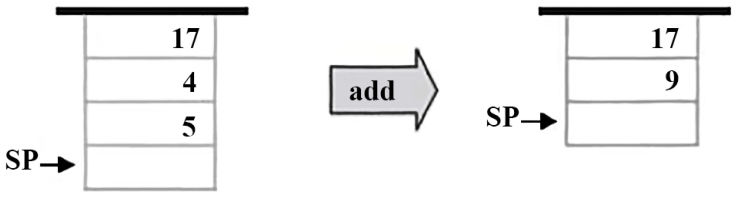
4.2.2 Struktura ukazov in programa

Program, ki se izvaja na navideznem stroju, je zbirka ene ali več datotek s končnico `.vm`, katera vsebuje eno ali več funkcij. Vsak ukaz se nahaja v svoji vrstici in je v enem izmed naslednjih formatov: `ukaz` (primer: `add`), `ukaz arg` (primer: `goto loop`), `ukaz arg1 arg2` (primer: `push local 3`).

Ob zagonu računalnika Hack ta nastavi programski števec na 0, kar pomeni, da mora biti v ukaznem pomnilniku v registru z naslovom 0 koda, ki poskrbi za uspešen zagon računalnika. Navidezni stroj zato najprej določi na-



Slika 4.3: Stanje sklada pri klicu nove funkcije.



Slika 4.4: Primer ukaza `add`, ki sešteje nazadnje vstavljena operanda v skladu.

slov registra, v katerem bo začetek sklada, in nato pokliče funkcijo `Sys.init`, ki poskrbi za inicializacijo operacijskega sistema in klicanje glavne metode programa (Main.main()), ter po zaključku izvajanja programa vstopi v neskončno zanko.

4.3 Prevajalnik

Za izvajanje Jack programov na navideznem stroju je potrebno kodo prevesti v vmesni jezik. Za to poskrbi program, ki se mu reče prevajalnik (ang. compiler). Prevajanje je sestavljeno iz analize sintakse jezika in generiranja kode, razumljive navideznemu stroju.

4.3.1 Programski jezik Jack

Jack je objektno usmerjen programski jezik, podoben Javi, a je v primerjavi z njo zelo enostaven. Ima le tri podatkovne tipe (`int, string, boolean`), ki so dolgi 16 bitov. Ne omogoča dedovanja, kar pomeni, da se metode nekega razreda zagotovo nahajajo v njem in za njih ni potrebno gledati višje v razrede, iz katerih bi bil lahko trenutni razred dedovan. Prav tako ni možno uporabljati javnih spremenljivk, ampak je za sklicevanje na njih potrebno implementirati potrebne metode. Manjkata tudi `switch` in `for` zanka.

Vse te pomankljivosti jezika naredijo implementacijo prevajalnika veliko enostavnejšo. K temu pripomore tudi naiven pristop prevajanja, katerega bi se dalo optimizirati na način, da bi prevajalnik zaznal operacijo, ki jo prevaja, ter jo poizkusil implementirati s čim manj ukazi.

4.3.2 Analiza sintakse

Analiza sintakse se deli na leksikalno in sintaksno analizo jezika. Glavna naloga analize je razumevanje strukture programa glede na slovnico jezika Jack. Tako lahko prevajalnik prepozna začetek in konec razreda ali metode,

```
class Square {

   field int x, y; // zgronji levi kot kvadrata
   field int size; // velikost kvadrata v pikslih
         .
         .
         .
```

Slika 4.5: Primer razreda Jack, ki je leksikalno analiziran na Sliki 4.6.

```
<tokens>
<keyword> class </keyword>
<identifier> Square </identifier>
<symbol> { </symbol>
<keyword> field </keyword>
<keyword> int </keyword>
<identifier> x </identifier>
<symbol> , </symbol>
<identifier> y </identifier>
<symbol> ; </symbol>
<keyword> field </keyword>
<keyword> int </keyword>
<identifier> size </identifier>
<symbol> ; </symbol>
         .
         .
         .
```

Slika 4.6: Primer leksikalno analiziranega razreda Jack, prikazanega na Sliki 4.5.

deklaracijo spremenljivk, izraze in njihove oblike ter vsa ostala sintaktična pravila jezika.

Pri leksikalni analizi gre za razbitje kode na najmanjše dele, določene s strani jezika, katerega prevajamo. Prav tako se odstranijo komentarji in presledki, ki so za prevajalnik nepomembni. Pri leksikalni analizi je vsak atom (najmanjši element jezika) izpisan v svoji vrstici v xml datoteko, kjer je obdan z oznako, ki opisuje njegovo identiteto. Ta je lahko neka številska konstanta, simbol, spremenljivka, izraz ali pa kateri izmed ostalih elementov slovnice jezika, ki ga prevajamo. Primer leksikalne analize Jack razreda iz Slike 4.5 se nahaja na Sliki 4.6.

```
<class>
  <keyword> class </keyword>
  <identifier> Square </identifier>
  <symbol> { </symbol>
  <classVarDec>
    <keyword> field </keyword>
    <keyword> int </keyword>
    <identifier> x </identifier>
    <symbol> , </symbol>
    <identifier> y </identifier>
    <symbol> ; </symbol>
  </classVarDec>
  <classVarDec>
    <keyword> field </keyword>
    <keyword> int </keyword>
    <identifier> size </identifier>
    <symbol> ; </symbol>
  </classVarDec>
                .
                .
                .
```

Slika 4.7: Primer sintaksno analiziranega Jack razreda iz Slike 4.5.

Po leksikalni analizi je potrebno kodo sestaviti v smiselne segmente, kot narekuje slovnica, iz katerih se bo nato generirala vmesna koda. Tako sintaksno analizirana koda dobi tudi drevesno strukturo, ki je lepše berljiva, kot je razvidno iz Slike 4.7. V tem koraku se tudi testira, ali je arhitektura prevajalnika pravilno zastavljena pred samim generiranjem kode.

Celoten postopek analize sintakse temelji na rekurzivnem branju kode, katera mora slediti slovničnim pravilom izraza, v katerem se trenutno nahaja. Dobro napisan prevajalnik tako omogoča tudi boljše zaznavanje napak med prevajanjem, saj izpiše mesto in vzrok napake.

4.3.3 Generiranje kode

Po razčlenitvi kode na posamezne segmente je te potrebno pretvoriti v vmesno kodo. Prvi korak je shranjevanje spremenljivk v podatkovno strukturo HashMap, kjer je za vsako vrsto spremenljivk potrebno narediti svoj HashMap. Iz posameznega HashMapa se pobriše vse spremenljivke, ko se zaključi področje, v katerem so bile uporabljene. Za ključ je shranjeno ime

```
1  class Main {
2
3     function void main() {
4        do Output.printInt(1 + (2 * 3));
5        return;
6     }
7
8  }
```

Slika 4.8: Primer Jack razreda, ki na zaslon izpiše število 7.

```
1  function Main.main 0
2  push constant 1
3  push constant 2
4  push constant 3
5  call Math.multiply 2
6  add
7  call Output.printInt 1
8  pop temp 0
9  push constant 0
10 return
```

Slika 4.9: Primer vmesne kode Jack razreda iz Slike 4.8.

spremenljivke, kot vrednost pa tabela, v kateri se nahaja tip spremenljivke in pa njena zaporedna številka v tej vrsti spremenljivk, katera nam pove mesto, kjer se ta spremenljivka nahaja v pomnilniku.

Pri generiranju kode se za vsako funkcijo generira oznaka z imenom razreda, kateremu sledi ime funkcije, da nato navidezni stroj ve, kje se ob klicu na to funkcijo nahaja koda, ki jo je potrebno izvršiti. Prav tako se morajo unikatne oznake generirat za posamezno `while` zanko in `if` stavek.

Generiranje kode pri izrazih poteka z rekurzivnim obratnim obhodom skozi kodo, katera je bila ustvarjena pri analizi sintakse. Takšen prehod skozi kodo omogoča vstavljanje operandov na sklad pred operacijo, ki se bo izvršila nad podatki. Primer vmesne kode, ki se je generirala iz Jack razreda na Sliki 4.8, se nahaja na Sliki 4.9 .

4.4 Operacijski sistem

Operacijski sistem je program (ali več programov), ki v računalniku skrbi za lažje povezovanje strojne opreme s programsko opremo. Njegova vloga je skrbeti za širok spekter operacij, katere med drugim omogočajo prikaz slike na zaslonu, uporabo tipkovnice, upravljanje s pomnilnikom, uporabo knjižnic za delo z matematičnimi funkcijami in nizi, itd.

Operacijski sistem Jack teče na računalniku Hack in je zelo minimalističen, saj vsebuje le nekaj osnovnih servisov. Industrijski operacijski sistemi poleg veliko bolj izpopolnjenih servisov vsebujejo še sistem za upravljanje z diskom, delo s procesi, skrbijo za varnost in še veliko več. Operacijski sistem Jack je napisan v Jack programskemu jeziku in preveden v binarno kodo kot ostali programi.

4.4.1 Standardne knjižnice Jack jezika

Programski jezik Jack vsebuje osem standardnih razredov: `Math`, `Memory`, `Screen`, `Output`, `Keyboard`, `String`, `Array` in `Sys`. Te so na platformi Hack predstavljene kot operacijski sistem, saj so to osnovni moduli, ki se pojavljajo v večini sodobnih operacijskih sistemov. Prav tako pa se pojavljajo v standardnih knjižnicah sodobnih programskih jezikov.

Posamezen modul operacijskega sistema Jack smo razvili in testirali ločeno od drugih modulov. To nam je omogočalo izgradnjo posameznega modula še pred samo implementacijo modulov, od katerih je bil ta odvisen. Na tak način se je razvil tudi operacijski sistem Linux, kjer so posamezen modul testirali s pomočjo modulov v operacijskem sistemu Unix. V našem primeru pa sta že avtorja knjige [1] implementirala vse module, katere smo nato uporabili pri implementaciji posameznega modula.

4.4.2 Opis razredov operacijskega sistema Jack

Posamezen razred igra pomembno vlogo pri hitrosti delovanja računalnika, saj so nekatere funkcije operacijskega sistema uporabljene zelo pogosto. Tako

jih je potrebno implementirati s čim hitrejšimi algoritmi. Implementacija razredov v Jack operacijskem sistemu je sledeča:

- razred **Math** vsebuje nekaj pogosto uporabljenih matematičnih funkcij.
- razred **Memory** skrbi za pravilno dodeljevanje in sprostitev pomnilnika v RAMu.
- razred **Screen** omogoča prikaz slike na zaslon. Vsebuje funkcije, ki omogočajo vklop ali izklop posame pike, izris črte, trikotnika in kroga.
- razred **Output** prikazuje besedilo na zaslon. Prikazanih je lahko največ 23 vrstic po 64 znakov. En znak tako zasede 8 pik v širino in 11 pik v višino.
- razred **Keyboard** omogoča branje pomnilniške mape tipkovnice ter tako vrača znak trenutno uporabljene tipke.
- razred **String** predstavlja niz znakov in vsebuje metode za delo z njim. Te so spreminjanje posameznega znaka v nizu, dodajanje in odvzemanje zadnjega znaka in ostale osnovne operacije za delo z nizi.
- razred **Array** vsebuje metodo za kreiranje nove tabele za določeno velikost ter metodo, ki to tabelo tudi pobriše po njeni končani uporabi.
- razred **Sys** poskrbi za zagon operacijskega sistema in za začetek izvajanja trenutnega programa.

Poglavje 5

Razširitev CPE

Množenje in deljenje sta v knjigi [1] implementirana na plasti operacijskega sistema, kar ju naredi nekoliko enostavnejša za implementacijo, a hkrati tudi počasnejša pri njunem izvajanju. Odločili smo se, da ju implementiramo še na ravni strojne opreme, kar posledično vodi do razširitve CPE, ki bo sedaj podpirala tudi množenje in deljenje.

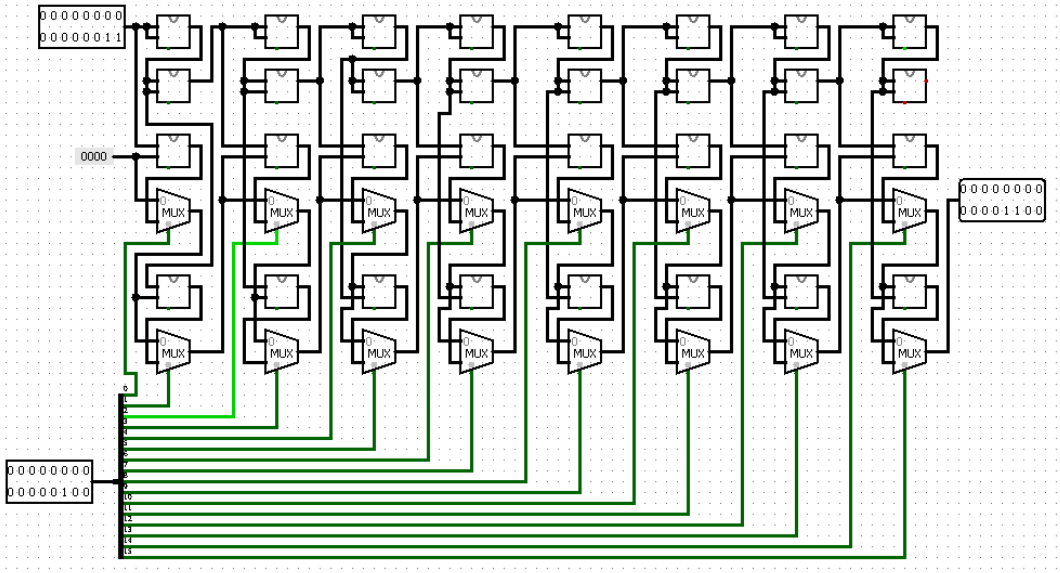
Za realizacijo tega je bilo najprej potrebno razširiti ALE z možnostjo izvajanja dveh novih operacij. Poleg obstoječih operacij na Sliki 3.2 je bilo potrebno poiskati dve novi kombinaciji ničel in enic kontrolnih bitov, ki omogočata izvedbo množenja in deljenja. Tako smo prišli do zaporedij, prikazanih na Sliki 5.1. Samega deljenja z vezjem nismo implementirali, je pa pravilnostna tabela ALE razširjena tako, da se lahko ob kasnejši implemen-

Tabela 5.1: Resničnostna tabela ALE za množenje in deljenje.

zx	**nx**	**zy**	**ny**	**f**	**no**	**izhod**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`
0	0	0	0	0	1	x*y
0	0	0	0	1	1	x/y

```
1  multiply(x,y):
2      sum=0;
3      shiftedX=x;
4      for j=0...(n-1) do
5          if(j-th bit of y)=1 then
6              sum=sum+shiftedX;
7          shiftedX=shiftedX+shiftedX;
```

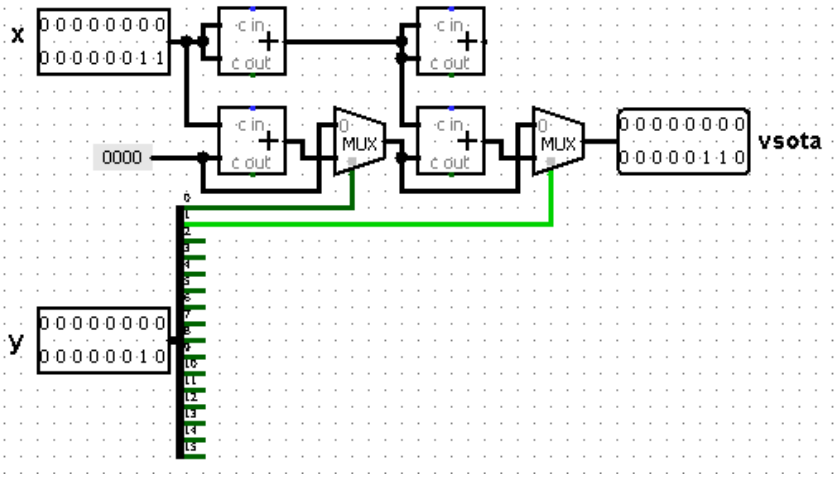
Slika 5.1: Psevdo koda množenja, implementiranega v ALE.



Slika 5.2: Vezje, ki zmnoži dve števili.

tacijo enostavno vključi tudi to, saj je ukaz, ki ga bi izvršil, že v izvajanju.

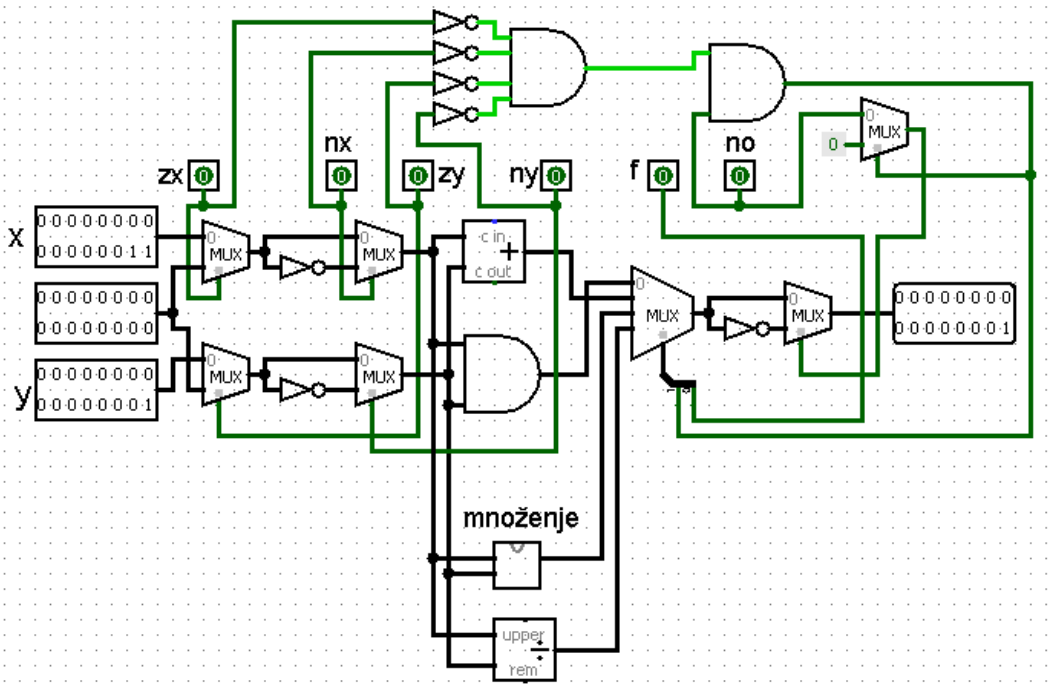
Algoritem množenja, ki smo ga implementirali, je hitrosti O(n), kjer je n število bitov, potrebnih za predstavitev števila v računalniku. V našem primeru ima n največjo vrednost 16, ker Hack operira s 16-bitnimi števili. Tako se operacija množenja v računalniku Hack izvrši v 16 iteracijah, ne glede na velikost števil, katera množimo. Psevdo-koda algoritma, ki se izvaja, je predstavljena na Sliki 5.1. V vsaki iteraciji se zgodi pomik bitov števila x v levo ter nato prišteje vsoti, kadar je n-ti bit števila y, kjer je n zaporedno število iteracij, nastavljeno na 1. Ta algoritem je z vezjem predstavljen na



Slika 5.3: Prvi dve iteraciji množenja iz Slike 5.2.

Sliki 5.2. Za boljšo preglednost pa sta na Sliki 5.3 prikazani prvi dve iteraciji množenja. Najprej se zgodi pomik bitov števila x v levo, kar se nato sešteje s trenutno vsoto. V multipleksorju pa se nato glede na n-to vrednost bita v številu y zgodi odločitev, ali je vsota enaka vsoti iz prejšnje iteracije, ali pa se upošteva nova.

Vključitev množenja v ALE je prikazana na vezju, ki se nahaja na Sliki 5.4. Vezje deluje tako, da z bitom `f` izbiramo med seštevanjem in operacijo `IN`, kadar je bit `no` nastavljen na 0. Ko pa je ta bit nastavljen na 1, z bitom `f` izbiramo med množenjem in deljenjem. Pri taki razširitvi bit `no` nekoliko izgubi pomen, saj naj bi ta ob vrednosti 1 negiral izhod, a ga v našem primeru pri množenju in deljenju ne. ALE bi bilo možno razširiti še tako, da bi zastavico `f` razširili na širino dveh bitov in tako v multipleksorju brez dodatnega vezja izbirali med štirimi funkcijami. Za takšno implementacijo bi ALE moral uporabljati 7 kontrolnih bitov namesto 6, ki jih uporablja sedaj. Za uporabljeno implementacijo smo se odločili zaradi nekoliko lažje kasnejše razširitve zbirnika, saj v njem ni potrebno dodajati dodatnega bita vsem ukazom, ampak se le doda dva nova ukaza. Prav tako takšna razširitev omogoča izvajanje vseh programov, ki so sprogramirani ali prevedeni za izvajanje na



Slika 5.4: Razširjena ALE, ki podpira tudi množenje in deljenje.

ALE, ki ni razširjena.

Z implementacijo množenja na nivoju CPE za njegovo izvajanje v zbirnem jeziku potrebujemo le en ukaz, kar ga naredi hitrejšega, saj ob implementaciji množenja z istim algoritmom na nivoju operacijskega sistema ta v zbirnem jeziku zasede 421 ukazov. Slaba stran čisto kombinacijskega množilnika pa je podaljšan urin cikel, ki posledično upočasni ostale, časovno manj zahtevne operacije. Čas množenja bi lahko zmanjšali s cevovodnim izvajanjem in seštevanjem delnih produktov.

Poglavje 6

Sklepne ugotovitve

V sklopu diplomskega dela je prikazan razvoj računalnika, na katerem je mogoča uporaba visokonivojskega programskega jezika. Skozi proces izdelave se srečamo z izgradnjo aritmetično-logične enote, ki je s še nekaj registri in programskim števcem del centralne procesne enote. Razumevanje tega omogoča razumevanje računalnika na nivoju strojnega jezika, kar omogoča načrtovanje zbirnega jezika, navideznega stroja in visokonivojskega programskega jezika. Izdelava računalnika Hack tako v primerjavi z izgradnjo industrijskega računalnika na enostavnejši način prikaže vlogo ter povezave različnih sklopov računalništva. Razumevanje teh elementov računalništva pa omogoča boljše in bolj kakovostno pisanje kode ter omogoča programerjem z razumevanjem tega večjo prednost na trgu.

Z vsemi področji sem se srečal že skozi študij na fakulteti, a mi je po končanem diplomskem delu postala njihova medsebojna povezava bolj jasna. Menim, da je bil to dober korak k izboljšanju mojih programerskih sposobnosti in k razumevanju v industriji uporabljenih tehnologij.

Na izdelanem računalniku bi bilo mogoče implementirati še veliko razširitev. Nekatere izmed njih so večje število vhodno-izhodnih naprav, razširitev programskih knjižnic jezika Jack, pohitritev nekaterih algoritmov ter razširitev aritmetično-logične enote, kar bi omogočalo hitrejše delovanje.

Literatura

[1] N. Nisan, S. Schocken. *The Elements of Computing Systems*. The MIT Press, 2008.

[2] Build a Modern Computer from First Principles. [Online]. Dosegljivo: http://nand2tetris.org/. [Dostopano 15. 9. 2017]

[3] Build a Modern Computer from First Principles: Nand to Tetris. [Online]. Dosegljivo: https://www.coursera.org/learn/build-a-computer/home/welcome. [Dostopano 15. 9. 2017]

[4] Build a Modern Computer from First Principles: Nand to Tetris Part 2. [Online]. Dosegljivo: https://www.coursera.org/learn/nand2tetris2/home/welcome. [Dostopano 15. 9. 2017]

[5] Jezik za opis strojne opreme. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Hardware_description_language. [Dostopano 15. 9. 2017]

[6] Navidezni stroj. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Virtual_machine. [Dostopano 15. 9. 2017]

[7] Obratni obhod drevesa. [Online]. Dosegljivo: http://wiki.fmf.uni-lj.si/wiki/Obratni_pregled. [Dostopano 15. 9. 2017]

[8] Logisim dokumentacija. [Online]. Dosegljivo:
http://www.cburch.com/logisim/docs.html. [Dostopano 15. 9. 2017]